Chapter 1
On RAM priority queues*

Mikkel Thorup†

Abstract

Priority queues are some of the most fundamental data structures. They are used directly for, say, task scheduling in operating systems. Moreover, they are essential to greedy algorithms. We study the complexity of priority queue operations on a RAM with arbitrary word size. We present exponential improvements over previous bounds, and we show tight relations to sorting.

Our first result is a RAM priority queue supporting `insert` and `extract-min` operations in worst case time $O(\log\log n)$ where n is the current number of keys in the queue. This is an exponential improvement over the $O(\sqrt{\log n})$ bound of Fredman and Willard from STOC'90. Our algorithm is simple, and it only uses AC^0 operations, meaning that there is no hidden time dependency on the word size. Plugging this priority queue into Dijkstra's algorithm gives an $O(m\log\log m)$ algorithm for the single source shortest path problem on a graph with m edges, as compared with the previous $O(m\sqrt{\log m})$ bound based on Fredman and Willard's priority queue.

Our second result is a general equivalence between sorting and priority queues. A priority queue is *monotone* if the the minimum is non-decreasing over time, as in greedy algorithms. We show that on a RAM the amortized operation cost of a monotone priority queue is equivalent to the per key cost of sorting. For example the equivalence implies that the single source shortest path problem on a graph with m edges is no harder than that of sorting m keys. With the current RAM sorting, this gives an $O(m\log\log m)$ time bound, as above, but the relation holds no matter the future developments in RAM sorting.

From the equivalence result, for any fixed $\varepsilon > 0$, we derive a monotone $O(\sqrt{\log n}^{1+\varepsilon})$ priority queue with constant time `decrease-key`. Plugging this into Dijkstra's algorithm gives an $O(n\sqrt{\log n}^{1+\varepsilon} + m)$ algorithm for the single source shortest path problem on a graph with n nodes and m edges, complementing the above $O(m\log\log m)$ algorithm if $m \gg n$. This improves the $O(n\log n/\log\log n + m)$ bound by Fredman and Willard from FOCS'90 based on their $O(\log n/\log\log n)$ priority queue with constant `decrease-key`.

1 Introduction

Priority queues are some of the most fundamental data structures. A *priority queue* is a dynamic a representation of a set, or multiset, X of keys. In its most basic form, a priority queue supports the operations `insert`(x) setting $X := X \cup \{x\}$, and `extract-min` returning and deleting the smallest key from X. Also, we will refer to the operations: `find-min` returning $\min X$, `delete`(x) setting $X := X \setminus \{x\}$, and `decrease-key`(x, d) setting $X := (X \setminus \{x\}) \cup \{x - d\}$. Priority queues are used directly for, say, task scheduling in operating systems. Moreover, they are essential to greedy algorithms. We study the complexity of priority queue operations on a RAM with arbitrary word size. It is assumed, however, that each key is an integer contained in one (or any constant number of) word(s) - so that operations like comparisons can be carried out in constant time. We present exponential improvements over previous bounds, and we show tight relations to sorting.

In connection with their $(n\log\log n)$ RAM sorting algorithm, Andersson, Hagerup, Nilsson, and Raman [4] point out that the previous fastest $O(n\sqrt{\log n})$ sorting technique by Fredman and Willard [11] has applications within dynamic data structures, to which the $O(n\log\log n)$ sorting technique does not apply. To this end we present

THEOREM 1.1. *There is a RAM priority queue supporting* `insert` *and* `extract-min` *in* $O(\log\log n)$ *worst case time, where* n *is the current number of keys in the queue.*

Thus our operation cost matches the per key cost of sorting from [4]. Note that with tabulated multiplicity, we get `delete`, and hence `decrease-key`, in $O(\log\log n)$ amortized time, simply be skipping extracted keys of multiplicity 0. Also we can get `find-min` in constant time if we always remember the last extracted key and consider it part of the queue. The priority queue of Theorem 1.1 requires sparse tables with constant access time. With hash tables, and hence randomization, we only need linear space. If, instead, we use tries, we get a deterministic space bound of $O(nu^{\varepsilon})$, or $O(n + u^{\varepsilon})$ if we are satisfied with amortized time bounds. Here $u = 2^w$ is the size of the universe, and ε is any positive constant. Our algorithm is simple, and for the deterministic version, we only need an instruction set with comparisons, addition, subtraction, bitwise

*Part of this research was done while the author visited DIMACS.

†Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark; `mthorup@diku.dk`, `http://www.diku.dk/~mthorup`

'and' and 'or', and unrestricted shift to the left and to the right within a word. All these operations are in AC^0, meaning that we do not have any hidden time dependency on the word size.

Our result is an exponential improvement over the previous fastest priority queues by Fredman and Willard [11], supporting `insert` and `extract-min` in $O(\sqrt{\log n})$ time with a similar trade-off between hashing/randomization and space as ours. Also, in [11] they get an $O(\log n/\log\log n)$ time bound in deterministic linear space. The techniques from [11] are complicated and use operations like multiplication outside AC^0 [7]. However, in a recent paper [3], Anderson presents a simple deterministic priority queue achieving the $O(\sqrt{\log n})$ time bound, using only AC^0 operations and with space bounds like ours. For completeness it should be noted that with tabulated multiplicity, all keys in the queue are distinct, so $n \leq u$, and hence our priority queue is always at least as fast as van Emde Boas's $O(\log\log u)$ priority queue [24,25].

In contrast to our new priority queues, the priority queues from [3,11] are general dynamic search structures supporting $\texttt{find}(x)$ within the same time bounds. Here $\texttt{find}(x)$ finds the smallest key in the queue greater than or equal to x. Interestingly, it is provably impossible to extend our $O(\log\log n)$ priority queue with such a `find` operation. Miltersen has pointed out that setting $n = 2^{\log^{3/2} w}$, the arguments in [20,21] imply an $\Omega(\log^{1/3-o(1)} n)$ lower bound for `find`. This lower bound holds even if we allow randomization and amortization. The lower bound is, however, beaten both if the word size w is sufficiently long compared with n [2], or if it is sufficiently short [24,25]. In [20] it is asked if the lower bounds for `find` can be beaten if we restrict ourselves to `insert` and `extract-min`. Our new $O(\log\log n)$ upper bound for all word sizes answers this question in the affirmative. In particular this implies a separation between priority queue operations in the RAM model contrasting the comparison model were the cost per operations is $\Theta(\log n)$ both with and without `find`.

Plugging the priority queue of Theorem 1.1 into Dijkstra's algorithm [10], we get

Corollary 1.1. *There is an $O(m\log\log m)$ algorithm for the single source shortest path problem on a graph with m edges.*

Above Dijkstra's algorithm should only be taken as a representative for greedy algorithms. We could equally well apply our priority queue to the algorithm from [18], and get an $O(n\log\log n)$ algorithm for the alphabetic tree problem on n codes. Based on the priority queues from [12], the previous best bounds for the two problems were $O(m\sqrt{\log m})$ and $O(n\sqrt{\log n})$, respectively. The advantage to focusing on Dijkstra's algorithm is that it facilitates later discussions of constant time `decrease-key` operations.

The $O(\log\log n)$ bound in Theorem 1.1 matches the per key cost of the recent $O(n\log\log n)$ sorting algorithm by Andersson, Hagerup, Nilsson, and Raman [4], but will such a match be possible for all future developments in RAM sorting? The complexity of searching followed the per key cost of sorting in Fredman and Willard's fusion trees [11], but as we discussed above they are now provably left behind. Will the same happen to priority queues? Below we give a partial answer to this question.

By a *monotone* priority queue, we mean a priority queue with the restriction on $\texttt{insert}(x)$ that $x > \min X$. Thus, for a monotone priority queue, the minimum is non-decreasing over time. For contrast, we will often refer to standard priority queues without the monotonicity restriction as *unrestricted*. The monotonicity restriction on priority queues has been applied previously in [1]. Monotonicity would be prohibitive for priority queues used for, say, task scheduling in operating system, but it is not a problem for greedy algorithms like Dijkstra's shortest path algorithm. Our second result is a general equivalence on RAMs between sorting and monotone priority queues.

Theorem 1.2. *For a RAM with arbitrary word size, if we can sort n keys in time $n\cdot s(n)$, where s is non-decreasing, then (and only then) there is a monotone priority queue with capacity for n keys, supporting* `find-min` *in constant time and* `insert` *and* `delete` *in $s(n)+O(1)$ amortized time. The equivalence holds even if n is limited in terms of the word size w.*

To prove Theorem 1.2 we will show how to maintain a monotone priority queue spending constant time on `find-min` and constant amortized time on `insert` and `delete` plus some sorting time. Each key passing through the queue participates in at most one sorting, and when we sort a set of keys, they are all currently contained in the queue. As an immediate consequence of Theorem 1.2, we get the following strengthening of Corollary 1.1:

Corollary 1.2. *On a RAM, the single source shortest path problem on a graph with m edges is no harder than that of sorting m keys, no matter the future developments in sorting.*

Note that potentially Theorem 1.2 could be used to obtain lower bounds for sorting. As mentioned above, [20,21] imply amortized non-constant lower bounds on search operations. Similar non-constant lower bounds on monotone priority queue operations would imply a non-linear lower bound for RAM sorting.

We will now show how Theorem 1.2 can be used to construct priority queues with a constant time `decrease-key` operation. From [4] we know that we can sort in linear expected time if $(\log n)^{2+\varepsilon} \le w$ for some fixed $\varepsilon > 0$. Thus, by Theorem 1.2,

COROLLARY 1.3. *There is a monotone priority queue supporting* `find-min` *in constant time and* `insert` *and* `delete` *in expected constant amortized time if* $(\log n)^{2+\varepsilon} \le w$ *for some fixed* $\varepsilon > 0$.

By an *atomic* priority queue we refer to a priority queue where all the above operations are done in constant amortized time, as in Corollary 1.3.

From [12], essentially we have (for details, see Appendix B or [22, §4])

LEMMA 1.1. *Provided monotone/unrestricted atomic priority queues for up to* $f(n)$ *keys, there is a monotone/unrestricted priority queue with capacity* n*, supporting the operations* `find-min`, `insert`, *and* `decrease-key` *in constant amortized time, and* `delete` *in time* $O(\log n/\log f(n))$.

In [12] they provide unrestricted atomic priority queues for up to $(\log n)^2$ keys, thus getting an unrestricted priority queue with capacity n, supporting `find-min`, `insert`, and `decrease-key` in constant amortized time, and supporting `delete` in $O(\log n/\log(\log n)^2) = O(\log n/\log\log n)$ amortized time. With tabulated multiplicity, $w \ge \log n$, so Corollary 1.3 implies a monotone atomic priority queues for up $\exp(\sqrt{\log n}^{1-\varepsilon'})$ keys for any fixed $\varepsilon' > 0$. Thus we get `delete` in time $O(\log n/\log(\exp(\sqrt{\log n}^{1-\varepsilon'}))) = O(\sqrt{\log n}^{1+\varepsilon'})$, that is,

COROLLARY 1.4. *There is a monotone priority queue with capacity* n *supporting* `find-min`, `insert`, *and* `decrease-key` *in expected amortized constant time, and* `delete` *in expected amortized time* $O(\sqrt{\log n}^{1+\varepsilon'})$ *for any fixed* $\varepsilon' > 0$.

Plugging Corollary 1.4 into Dijkstra's algorithm [10], we get

COROLLARY 1.5. *For any* $\varepsilon' > 0$*, we can solve the single source shortest path problem on a graph with* n *nodes and* m *edges in expected time* $O(n\sqrt{\log n}^{1+\varepsilon'} + m)$.

The previous best bound linear in m was $O(n \log n/\log\log n + m)$, based on the priority queues from [12]. For completeness, Corollary 1.5 should also be compared with the word size dependent bound of $O(n\sqrt{w} + m)$ from [1]. In fact, the bound from [1] is based on priority queues restricted beyond monotonicity, tailored particularly for Dijkstra's algorithm. Since $w \ge \log n$ this bound is only better for $w = (\log n)^{1+o(1)}$. Recall that for small m, we have the complementing $O(n + m\log\log m)$ bound from Theorem 1.1.

The above time bounds hold for integer keys occupying any constant number of words. If there is no bound on the number of words per integer, we need to add the time it takes to read the distinguishing prefixes [5]. As discussed in [4], multiple word integers is not an exotic special case. For example, the IEEE 754 floating-point standard is designed so that the ordering of floating point numbers can be deduced by perceiving their representations as multiple word integers [15].

The priority queue of Theorem 1.1 is presented in Section 2. To show that this priority queue is simple enough to be of practical relevance, when possible, we will complement references to known general constructions by simpler direct constructions. The reduction to sorting from Theorem 1.2 is presented in Section 3.

2 An $O(\log\log n)$ priority queue

2.1 A priority queue for small integers. The goal of this section is to prove

THEOREM 2.1. *There is a priority queue with up to* n *keys, supporting* `insert` *and* `extract-min` *operations of* $(w/\log n)$*-bit integers in* $O(\log\log n)$ *time.*

From [6], we need

LEMMA 2.1. *We can merge two sorted lists, each of at most* k *keys stored in a single word, into a single sorted list stored in two words in time* $O(\log k)$.

COROLLARY 2.1. *For* $n \ge k$*, given two lists of* n (w/k)*-bit integers, spread over* n/k *words, we can merge them into* $2n/k$ *words in time* $O(n/k \cdot \log k)$.

Corollary 2.1 is implicit in [6] but in a parallel setting. We note the following simple sequential derivation:

Proof. Pop the first word of each of the two lists to be merged. Apply Lemma 2.1 to these two words, obtaining w_0 and w_1. Now w_0 is the first word in the merged list while w_1 is pushed on the list from which its last/biggest key came from. Thus we repeat until one list is empty. Hence each application of Lemma 2.1 extracts one word for the final merged list while the other goes back for the next iteration. □

To proof Theorem 2.1, we will use Lemma 2.1 and Corollary 2.1 with $k = \log n$.

Proof of Theorem 2.1. Note that the complexity stated in Theorem 2.1 is in terms of the maximum number of keys in the priority queue. At the end of the next section, we will reduce this dependency to being on the current number of keys.

We will view the merging described in Corollary 2.1 as divided into n/k *steps*. Thus it takes as many steps to merge two lists as there are words in each list, and for $k = \log n$, each step takes time $O(\log\log n)$. Referring to our concrete proof of Corollary 2.1, each step essentially corresponds to two applications of Lemma 2.1.

Now, concerning `insert`, we will have a "reception" B_0 and "buffers" $B_1, \ldots, B_{\log n - \log\log n}$. The reception

B_0 is a sorted list of at most $\log n$ keys stored in a single word. Supposing that B_0 is not yet full, by Lemma 2.1, we can merge a single new key into B_0 in time $O(\log\log n)$.

The buffers B_i, $i > 0$, may contain 0, 1, or 2 sorted lists of $2^{i-1}\log n$ keys, each distributed over 2^{i-1} words. Hence, the maximum capacity of B_i is 2^i words. If B_i contains 2 lists these may be partially merged.

The `insert` operations are organized in rounds of length $\log n$. This is the time it takes us to fill B_0. Also a round allows us to do one merge step in each buffer. Within a round, we have an index running from $\log n - 1$ and down to 0. When the process starts, all B_i are empty, and $i = \log n - 1$.

ALGORITHM A. `insert`(x)

A.1. Insert x into B_0.

A.2. If $i = 0$,

A.2.1. Move the list of B_0 to B_1.

A.3. If $0 < i \leq \log n - \log\log n$ and B_i contains 2 lists,

A.3.1. Make one merge step on these two lists.

A.3.2. If the merging is thereby completed,

A.3.2.1. Move the resulting list to B_{i+1}.

A.4. $i := i - 1 \bmod \log n$.

Note that no buffer is visited if $i > \log n - \log\log n$. The point is that each buffer B_i is visited exactly once in each round. Clearly the `insert` runs correctly in $O(\log\log n)$ time if it is true that we never get more than 2 lists in any buffer. Thus we have to show for step A.3.2.1, that B_{i+1} contains at most one list. Now, B_{i+1} contains 2 lists for at most 2^i rounds, since this is the number of steps it takes to merge two lists of 2^i words each. Thus it suffices to show

CLAIM 1 *Buffer B_j is emptied at most every 2^jth round.*

Proof. The proof is by induction on j. The base case, $j = 0$, is immediate since B_0 is emptied once every round; namely when $i = 0$.

For the inductive step, we note that when the first list is moved to B_j, then B_{j-1} is emptied. Thus, by induction, it takes at least 2^{j-1} rounds before a second list is moved from B_{j-1} to B_j, and further 2^{j-1} rounds before the two lists are merged so that the result can be moved to B_{j+1}. $\diamond$

In order to implement `extract-min`, we associate a standard log-heap M [26] with the first/smallest key of each B_i. Since there are at most $\log n$ B_i, each operation on M is supported in $O(\log\log n)$ time.

Now `extract-min` is implemented as follows. First, using M, we find the buffer B_i with the smallest first key, hence the smallest overall key x. Second we add the next key of B_i to M—even if B_i contains two lists, this is easily organized locally within B_i in constant time. Assume that no keys are zero—zero keys can always be dealt with separately. Then, third, we zero x's position in L (so that it stays neutral in later merges. We do not yet want to start shifting keys around in the words). Finally, we return x. The first two operations are done in time $O(\log\log n)$, and the last two are done in constant time. Note that we resist the temptation to do any merge steps during `extract-min`; otherwise we would be at risk of violating the the counting from Claim 1.

In order to maintain M properly, we note that `insert` may change the first key of B_0. Also, when moving B_i to B_{i+1}, we need to delete from M the biggest smallest key in the two. Both cases are dealt with in time $O(\log\log n)$.

At the moment `extract-min` does not decrease the size of the B_i. We need to prevent the priority queue from growing beyond the order of the maximal number of non-zero keys. This is done simply by changing step A.3.2.1 to

A.3.2.1' If the first half of the keys in the resulting list are zeroed, remove them. Afterwards the list contain 2^i words, and stay as a single list in B_i. Otherwise, if the first half contains non-zero keys, move the full list of 2^{i+1} words to B_{i+1}.

Clearly this change does not affect our argument that each buffer contains at most two lists—it can only take longer time for a buffer to get emptied. As a consequence of our new step A.3.2.1', we will never start using buffer B_{i+1} unless we have at least $2^{i-1}\log n$ non-zero keys. Hence the used buffers never get a total capacity for more than 8 times as many keys as are needed.

Above, it might seem as if we need to know n in advance for the packing of keys in words. However, all the algorithm need to know in advance is the word size w an the number b of bits in each key. The time bound only requires that we have capacity for $\log n$ keys in each word, i.e. that $\log n \leq w/b$. This completes the proof of Theorem 2.1. $\square$

Elaborating on the above proof, it is possible to construct a $O(\log^{**} n)$ priority queue for integers with $(w/(\log n \log\log n))$ bits each. The immediate changes are to set $k = \log n \log\log n$ and to let M be a recursive version of the priority queue thus constructed. Unfortunately, B_0 becomes much more messy to implement. Note, for example, that already after one recursion, all keys fit into one word. Theorem 2.1, however, suffices for the proof of Theorem 1.1. Our real bottleneck is the range reduction to be presented in the next section.

It should be mentioned that merging previously has been used in connection with priority queues to make them adaptable to sequential storage [13]. Our algorithm seems simpler than the one in [13], so it could be interesting to see how it performed on sequential storage if we ignored the packing.

2.2 Priority queues for arbitrary integers. In this section we complete the proof of Theorem 1.1. To get from arbitrary integers to short integers we will use a recursive range reduction which can be seen as a simple specialized variant of van Emde Boas' data structure [24,25], inspired by the developments in [5,16,17,19]. Let $T(n, b)$ be the time for `insert` and `extract-min` in a priority queue with up to n b-bit integers. We assume that b but not n is known in advance. By Theorem 2.1, $T(n, w/\log n) = O(\log\log n)$. We will show that

$$T(n, b) = O(1) + T(n, b/2). \tag{2.1}$$

Here b is assumed to be a power of 2. For any b-bit integer x, let $\mathtt{low}(x)$ denote the contents of last $b/2$ bits, and let $\mathtt{high}(x)$ denote the contents of the first $b/2$ bits. Thus, $x = \mathtt{high}(x)2^{b/2} + \mathtt{low}(x)$.

Let H be a $(b/2)$-bit integer priority queue. H will be used for the `high`-values of the inserted keys. We will maintain a separate table with the values $h \in H$; either in linear space using random hash table [9,19], or for any fixed $\varepsilon > 0$, deterministically in $O(nu^\varepsilon)$ space using tries (for details, see Appendix A or [22, §4]). Moreover, for each $h \in H$, we will have an integer $l(h)$ and a $(b/2)$-bit integer priority queue $L(h)$ such that

- $\{l(h)\} \cup L(h) = \{\mathtt{low}(x) | x \in Q, \mathtt{high}(x) = h\}$ where Q is the set of current keys.
- $l(h) < \min L(h)$.

Besides, we will have an integer $h_{min} = \min H$. Initially $H = \emptyset$ and $h_{min} = \infty$. We are now ready to present concrete implementations of `insert` and `extract-min` settling recurrence (2.1).

Algorithm B. $\mathtt{insert}(x)$
B.1. $(h, l) := (\mathtt{high}(x), \mathtt{low}(x))$.
B.2. If $h \notin H$,
B.2.1. $h_{min} := \min(h, h_{min})$.
B.2.2. $\mathtt{insert}(h, H)$.
B.2.3. $l(h) := l$.
B.3. If $h \in H$,
B.3.1. $(l(h), l) := (\min(l(h), l), \max(l(h), l))$.
B.3.2. $\mathtt{insert}(l, L(h))$.

Algorithm C. `extract-min`
C.1. $x := h_{min} \cdot 2^{b/2} + l(h_{min})$.
C.2. If $|L(h_{min})| > 0$, $l(h_{min}) := \mathtt{extract\text{-}min}(L(h))$.
C.3. If $|L(h_{min})| = 0$, $h_{min} := \mathtt{extract\text{-}min}(H)$.
C.4. Return x.

Each of the above algorithms make exactly one recursive call, so this settles the time bound described in recurrence (2.1).

Proof of Theorem 1.1: Let w be the full word length. Starting with w-bit integers, we apply recurrence (2.1) $\log\log n$ times. At cost $O(\log\log n)$ per operation, this reduce our problem to dealing with $(w/\log n)$-bit integers. By Theorem 2.1, these can be dealt with in $O(\log\log n)$ time, so the total time is $O(\log\log n)$ per operation.

Above, we have not taken into account that we want our complexities in terms of the current number of keys in the queue. In order to achieve this, we will generally be working on a "current" queue Q with the current keys, as well as on two auxiliary queues Q^-, Q^+. The capacities of Q^-, Q, Q^+ will be $2^i, 2^{i+1}, 2^{i+2}$ for some i. Moreover, we will have $Q = Q^- \cup Q^+$. Finally, the current queue Q will always be at least 1/4 full. Consequently, our capacity is at most 14 times the needed capacity. The system is started with Q half full. We only use the system when we have at least 2 current keys, and when this happens, we let the capacities be $2, 4, 8$.

In connection with any query q. If Q is more than half full and Q^- is non-empty, first we call $x := \mathtt{extract\text{-}min}(Q^-)$; $\mathtt{insert}(x, Q^+)$, and then we perform q. If Q gets filled, by that time $Q^- = \emptyset$ and $Q^+ = Q$, so Q^+ is half full. Hence we can restart setting $(Q^-, Q, Q^+) := (Q, Q^+, \emptyset)$. If Q is less than half full and Q^+ is non-empty, first we call $x := \mathtt{extract\text{-}min}(Q^+)$; $\mathtt{insert}(x, Q^-)$ twice, and then we perform q. If Q becomes 1/4 full, by that time $Q^+ = \emptyset$ and $Q^- = Q$ is half full, so we can restart setting $(Q^-, Q, Q^+) := (\emptyset, Q^-, Q)$. Clearly this standard doubling/halving does not change our asymptotic complexities. □

3 Equivalence between sorting and priority queues

3.1 The general reduction. This section is devoted to the general proof of Theorem 1.2. For simplicity, we assume that our queues are always non-empty. Fix $k = \log n \log\log n$. We will assume:

(i) Sparse universal tables, either by hashing [9] in linear space with expected constant look-up time, or for any $\varepsilon > 0$, by tries in space $O(\varepsilon^{-1}u^\varepsilon)$ doing look-up by ε^{-1} comparisons (for details, see Appendix A or [22, §4]).

(ii) Unrestricted atomic priority queues for up to k keys. In [12] such queues are provided, which, in fact, take up to $(\log n)^2$ keys. They require $O(n)$ preprocessing time and space.

(iii) Unrestricted atomic priority queues for (w/k)-bit keys. Such queues will be presented in the next section. Really we should have written $\lceil w/k \rceil$-bit keys, but for ease of presentation, we generally ignore rounding problems as long as they only affect our results by constant factors (< 2). The case where $k > w$ will be taken special care of in the next section.

(iv) Your favorite sorting algorithm `sort`.

We will perceive words as divided into k fields of size w/k. For any key x, we let $x[i]$ denote the contents of the ith field of the word containing x. Also, $x[i..j] = x[i] \cdots x[j]$. For any two keys x, y, by $\mathtt{split}(x, y)$ we denote the first i such that $x[i] \neq y[i]$. If $x = y$, $\mathtt{split}(x, y) = k + 1$. Applying the "lead", "compress", and "msb" operations from [11, §5], we compute `split` in constant time. Below, instead of implementing `delete` directly, for simplicity, we will start by just allowing `delete-min`, deleting the smallest key. The general `delete` operation will be included at the end of this section.

Our priority queue consists of the following components:

- The smallest key μ in the queue. Thus `find-min` is trivially done in constant time. Note for any keys x, y in the queue that $\mathtt{split}(\mu, x) > \mathtt{split}(\mu, y)$ implies $x < y$.

- Some sorted lists $L_1, \ldots, L_k$ of keys where L_i satisfy $\forall x \in L_i : \mathtt{split}(\mu, x) > i$. Note that $i < j$ implies that any key in L_j could also have been in L_i. Also note that if we replace μ by μ' and $\mu \leq \mu' \leq \min L_i$, then this does not violate the condition on L_i.

  Using an unrestricted atomic priority queue from (ii), in constant amortized time, we can keep track of which of the list $L_1, \ldots, L_k$ that has smallest first key. This key is hence the smallest of all the keys in the lists. Thus $\min \bigcup_i L_i = \min_i \{\min L_i\}$ is computable in constant amortized time.

- From (iii) we get unrestricted atomic priority queues $A_1, \ldots, A_k$ for (w/k)-bit keys – corresponding to the fields of our original keys. For $i = 1, \ldots, k$, it is required that $\mu[i] < \min A_i$. With each $a \in A_i$ is associated a set $S_i(a)$ of keys x such that $x[1..i] = \mu[1..i-1]a$. For the membership in A_i and the entries to $S_i(\cdot)$, we use the universal tables from (i).

  Let $\overline{A_i}$ denote $\bigcup_{a \in A_i} S_i(a)$. Then $\forall x \in \overline{A_i} : \mathtt{split}(\mu, x) = i$, so $i > j$ implies $\max \overline{A_i} < \min \overline{A_j}$.

  We will also have an unrestricted atomic priority queue from (ii) keeping track of the maximum index i of a non-empty A_i.

Above $(\{\mu\}, L_1, \ldots, L_k, \overline{A_1}, \ldots, \overline{A_k})$ is understood to be a partitioning of the keys currently in the queue. We are now ready to implement `insert` and `delete-min`.

Algorithm D. $\mathtt{insert}(x)$

D.1. $i := \mathtt{split}(x, \mu)$.
D.2. If $x[i] \notin A_i$ then $\mathtt{insert}(x[i], A_i)$, setting $S_i(x[i]) := \emptyset$.
D.3. $S_i(x[i]) := S_i(x[i]) \cup \{x\}$.

Algorithm E. `delete-min`

E.1. $\mu_L := \min \bigcup_h L_h$.
E.2. $i := \mathtt{split}(\mu_L, \mu)$.
E.3. Let j be maximum such that A_j is non-empty.
E.4. $a := \min A_j$.
E.5. If $i < j$ or ($i = j$ and $\mu_L[i] \geq a$) then
E.5.1. $L_j := \mathtt{sort}(S_j(a))$.
E.5.2. Delete a from A_j.
E.5.3. $\mu_L := \min\{\mu_L, \min L_j\}$.
E.6. $\mu := \mu_L$.
E.7. Delete μ_L from its list.

To see the correctness of Algorithm E, in connection with the condition of step E.5, observe:

- If $i \leq j$, then $L_j = \emptyset$; for assume $L_j \neq \emptyset$. Then $\mathtt{split}(\min L_j, \mu) > j$, and hence $\min L_j < \mu_L$ contradicting the minimality of μ_L. Thus we do not loose any keys when overwriting L_j in step E.5.1.

- The condition implies that the coming minimum key in the queue has prefix $\mu[1..j-1]a$. Thus after step E.6, L_j will correctly satisfy that $x[1..j] = \mu[1..j]$ for all $x \in L_j$.

Above we maintain the priority queue spending constant amortized time per `insert` or `delete-min` operation plus sorting. Each key participate in exactly one sorting; namely when it is transferred from $\bigcup_i \overline{A_i}$ to $\bigcup_i L_i$. Thus `sort` is only called with *disjoint* subsets of the keys, and whenever $\mathtt{sort}(X)$ is called, X is a subset of the keys currently in the queue.

We will now augment our priority queue with the general `delete` operation. We will use a universal table $D \subseteq \bigcup_i L_i$ denoting keys that are to be deleted. Then `delete` may be implemented as follows.

Algorithm F. $\mathtt{delete}(x)$

F.1. If $x \neq \mu$ then
F.1.1. $i := \mathtt{split}(x, \mu)$
F.1.2. If $x[i] \in A_i$ and $x \in S_i(x[i])$ then
F.1.2.1. $S_i(x[i]) := S_i(x[i]) \setminus \{x\}$.
F.1.2.2. If $S_i(x[i]) = \emptyset$, delete $x[i]$ from A_i.
F.1.3. else
F.1.3.1. $D := D \cup \{x\}$.
F.2. else
F.2.1. $\mu_L := \min \bigcup_h L_h$.

F.2.2. While $\mu_L \in D$,
F.2.2.1. Delete μ_L from D and from its list L_h.
F.2.2.2. $\mu_L := \min \bigcup_h L_h$.
F.2.3. `delete-min`

For space reasons, if ever $|D| \geq |\bigcup_i L_i|/2$, we scan through all the L_i deleting the keys from D, and setting $D := \emptyset$ at the end. The cost of the scan is attributed to the `delete` operations that filled D. Thus still we maintain the priority queue spending constant amortized time per operation plus sorting. Sorting is only called on disjoint subsets of keys, and whenever `sort`(X) is called, X is a subset of the keys currently in the queue. This completes the proof of Theorem 1.2, assuming the unrestricted atomic priority queue from (iii), to be described in the next section.

3.2 An unrestricted atomic priority queue for small integers. As promised in (iii) in the previous, we will prove

THEOREM 3.1. *There is an unrestricted atomic priority queue with up to n $\lceil w/k \rceil$-bit keys, where $k = \log n \log\log n$, supporting* `find-min` *in constant time, and supporting* `insert`*, and* `delete` *in constant amortized time.*

If $w \leq k$, the keys have only 1 bit, so there are only two possible keys. In this case, it is therefore trivial to support an unrestricted atomic priority queue. Thus we may assume $w > k$, but then rounding will only affect our results by at most a factor 2, and hence rounding will be ignored for ease of presentation.

Our construction is a faster but amortized variant of the one used for Theorem 2.1 in Section 2.1. The keys will be stored in sorted lists $L_0, \ldots, L_{\log n}$ where either $L_i = \emptyset$, or $2^{i-1} < |L_i| \leq 2^i$. Moreover, if L_i is non-empty, it is distributed over at most $\lceil |L_i|/k \rceil$ words.

ALGORITHM G. `insert`(x)
G.1. $L' := (x)$.
G.2. $i := 0$.
G.3. While $L_i \neq \emptyset$ do
G.3.1. Merge L' and L_i into L'.
G.3.2. $L_i := \emptyset$.
G.3.3. $i := i + 1$.
G.4. $L_i := L'$.

In order to support `find-min` and `delete-min`, we use the unrestricted atomic priority queue from [12] to keep track of which of the at most $\log n$ lists that has the smallest first key. Let L_i be the list with the smallest first key x. Then `find-min` just returns x. To implement `delete-min`, we first delete x from L_i. If thereby $|L_i| \leq 2^{i-1}$, we empty L_i, reinserting the keys one by one using `insert`.

We will now argue that we spend only constant amortized time for each key inserted. If a key is reinserted by `delete-min`, we restart the accounting, attributing the cost to `delete-min`. Consider a particular key x. The merges (step G.3.1) that x participate in must have strictly increasing values of i. In a given merging, the lists contain between $2^{i-1}+1$ and 2^i keys. Loosing at most a factor 2, we assume that both lists are full, containing 2^i keys. Then, if $2^i \leq k$, the per key cost of the merging is $O(i/2^i)$. If $2^i > k$, the per key cost of the merging is $O(\log k/k)$. Thus, the maximum total cost per key is

$$\left(\sum_{i=1}^{\log k - 1} i/2^i + \sum_{i=\log k}^{\log n} \log k/k\right) = O(1).$$

Recall that `delete-min` has constant cost if we ignore the reinsertions. To account for the reinsertions, we use a potential function $\Pi = \sum_{L_i \neq \emptyset}(2^i - |L_i|)$ measuring the number of "missing" keys in the lists. Thus Π starts at 0, and generally $0 \leq \Pi \leq 2n$. Note that `insert` does not affect Π. The immediate consequence of `delete-min` is that we delete an key from a list, hence that we increase Π by one. Now, if L_i gets emptied, $|L_i| \leq 2^{i-1}$, so this decreases Π by $\geq 2^{i-1}$. Thus the total number of reinsertions is bounded from above by the total number of `delete-min` operations, implying that the amortized cost of `delete-min` is $O(1)$. To get general `delete`, as in the last section, we just introduce a table D over keys to be deleted, checking new minimums against D. This completes the proof of Theorem 3.1.

Acknowledgement

I wish to thank Arne Andersson, Amir Ben-Amram, Martin Farach, Peter Miltersen, Teresa Przytycka, and Jesper Träff for comments and discussions.

References

[1] R.K. AHUJA, K. MELHORN, J.B. ORLIN, AND R.E. TARJAN, Faster algorithms for the shortest path problem, *J. ACM* **37** (1990) 213–223.

[2] M. AJTAI, M. FREDMAN, AND J. KOMLÓS, Hash functions for priority queues, *Inf. Contr.* **63** (1984), 217–225.

[3] A. ANDERSSON, Sublogarithmic searching without multiplications. Technical report LU-CS-TR:95-146, University of Lund, 1995. To appear at FOCS'95.

[4] A. ANDERSSON, T. HAGERUP, S. NILSSON, AND R. RAMAN, Sorting in linear time? *in* "Proc. 27th STOC," pp. 427–436, 1995.

[5] A. ANDERSSON AND S. NILSSON, A new efficient radix sort , *in* "Proc. 35th FOCS," pp. 714–731, 1994.

[6] S. ALBERS AND T. HAGERUP, Improved parallel integer sorting without concurrent writing, *in* "Proc. 3rd SODA," pp. 463–472, 1992.

[7] P. Beame and J. Håstad, Optimal bounds on the decision problems on the CRCW PRAM, *J. ACM* **36** (1989), 643–670.

[8] A.M. Ben-Amram and Z. Galil, When can we sort in $o(n \log n)$ time?, *in* "Proc. 34th FOCS," pp. 538–546, 1993.

[9] J.L. Carter and M.N. Wegman, Universal classes of hash functions, *J. Comp. Syst. Sci.* **18** (1979), 143–154.

[10] E.W. Dijkstra, A note on two problems in connection with graphs, *Numer. Math.* **1** (1959), 269–271.

[11] M.L. Fredman and D.E. Willard, Surpassing the information theoretic bound with fusion trees, *J. Comp. Syst. Sc.* **47** (1993) 424–436.

[12] M.L. Fredman and D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *J. Comp. Syst. Sc.* **48** (1994) 533–551.

[13] M.J. Fischer and M.S. Paterson, Fishspear: a priority queue algorithm, *J. ACM* **41,** 1 (1994) 3–30.

[14] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34** (1987) 596–615. See also FOCS'84.

[15] J.L. Hennessy and D.A. Patterson, *Computer Organization and Design: the Hardware/Software Interface,* Morgan Kaufmann, San Mateo, CA, 1994.

[16] D.B. Johnson, A priority queue in which initialization and queue operations take $O(\log\log D)$ time, *Math. Syst. Th.* **15,** 4 (1982), 295–309.

[17] D. Kirkpatrick and S. Reisch, Upper bounds for sorting integers on random access machines, *Theor. Comp. Sci.* **28** (1984), 263–276.

[18] T.C. Hu and A.C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM J. Appl. Math.* **21** (1971), 514–532.

[19] K. Melhorn and S. Nähler, Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space, *Inf. Proc. Lett.* **35,** 4 (1990), 183–189.

[20] P.B. Miltersen, Lower bounds for union-split-find related problems on random access machines, *in* "Proc. 26th STOC," pp. 625–634, 1994.

[21] P.B. Miltersen, N. Nisan, S. Safra, and A. Wigderson, On data structures and asymmetric communication complexity. *in* "Proc. 27th STOC," pp. 103–111, 1995.

[22] M. Thorup, An $O(\log\log n)$ priority queue, Technical Report DIKU-TR-95-5, Department of Computer Science, University of Copehagen, 1995.

[23] M. Thorup, Equivalence between sorting and priority queues, DIMACS Technical Report 95-12, May 1995.

[24] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inf. Proc. Lett.* **6** (1977), 80–82.

[25] P. van Emde Boas, R. Kaas, and E. Zijlstra, Design and implementation of an efficient priority queue, *Math. Syst. Th.* **10** (1977), 99–127.

[26] J.W.J. Williams, Heapsort, *Comm. ACM* **7**, 5 (1964), 347–348.

A Machine models and space

For theoretical reasons it is worthwhile noting that hashing is not needed for the tabulation of H in the Section 2.2. Without hashing our time bounds do not only become deterministic, but also we avoid multiplication which is not in AC^0 [7]. Then, as for the sorting results [4], all word operations needed on the RAM, including the packed merging of Lemma 2.1 from [6], are in AC^0; that is, they can all be implemented by circuits of polynomial size and constant depth. More specifically, we need the following *restricted AC^0 instruction set*: comparisons, addition, subtraction, bitwise 'and' and 'or', and unrestricted shift to the left and to the right within a word. In fact it should suffice with shifts of lengths $w2^{-i}$, so we are only asking for $\log w$ shifts to be hardwired. Note that right shift is important in the sense that for the remaining operations alone, even if we include multiplication, we have an $\Omega(n \log n)$ lower bound for sorting [8].

Note that our Algorithm B implementing `insert` adds *one* new entry to *one* of our recursively defined H-tables. For our hash table solution, this implies that if we enumerate the tables as the they are allocated, and include the table numbers in our entries, we may confine ourselves to *one* universal hash table of size $O(n)$. Thus, with hashing the total space is $O(n)$. A similar observation is made in [19].

For the deterministic case, for any constant $\varepsilon > 0$, we can achieve the $O(\log\log n)$ time bound in space $O(nu^\varepsilon)$, where $u = 2^w$ is the size of the universe. This follows from the following result:

Proposition A.1. *On a RAM with restricted AC^0 instruction set, for any constant $\varepsilon > 0$, we can manage up to n universal tables with a total of up to n used entries, using constant deterministic access time and $O(nu^\varepsilon)$ space. Here $u = 2^w$ is the size of the universe and w is the word size.*

Suppose that RAMs have address length a constant fraction α of the word length. Compare with a traditional $O(n^{1+\beta})$ space algorithm. Set $\varepsilon = \alpha\beta/2$. Then $O(nu^\varepsilon)$ beats $O(n^{1+\beta})$ in terms of how small RAMs we can run on.

Proof. Consider each entry as a string of $1/\varepsilon$ characters of at most $w\varepsilon$ bits each. Each table is then implemented as an unordered trie of depth $1/\varepsilon$ where the children of a node are tabulated in "sub-tables" with $w\varepsilon$-bit "sub-entries". Since ε is a constant, the tries are accessed in constant time. Moreover, the total number of used sub-tables is $O(n)$ and each takes space $2^{w\varepsilon} = u^\varepsilon$. Thus, if we only allocate the sub-tables as we used them, the total space requirement is $O(nu^\varepsilon)$, as desired. □

The deterministic space bound can be improved to

$O(n+u^{\varepsilon})$ if we are satisfied with amortized $O(\log\log n)$ time bounds. We can then be lazy building up our tables/priority queues, only keeping the parts leading to the currently smallest key updated. The laziness alone allows us to operate in space $O(\sum_{i=0}^{\log\log n} 2^i 2^{w/2^{i+1}}) = O(2^{w/2}) = O(\sqrt{u})$. Inspired by [5,4,16], this should be combined with letting the top-level of the range reduction divide the words into pieces of $w\varepsilon$ bits each. Then the resulting table space becomes $O(u^{\varepsilon})$, as desired. Note that if we are not payed for unused space, then $O(n + u^{\varepsilon})$ is as good as linear on RAMs with address length a constant fraction of the word length.

B Getting constant decrease-key

In this section, we show how the "AF-heap" from [12, §2.2] gives a constructive proof of Lemma 1.1. For the unrestricted case, we just set $B = f(n)/\log n$. Then the AF-heap gives `delete` in time $O(\log n/\log B)$, but from [12, §2.3], we know that $f(n) = \Omega((\log n)^2)$, so $O(\log n/\log B) = O(\log n/\log f(n))$.

For the monotone case it should be noted that even if the resulting queue is monotone, the insertions in the AF-heap, do not all immediately respect monotonicity, but the exceptions fall in two groups:

- During the "ripple" operation, a minimum x is deleted and a replaced by a larger key y. In order to preserve monotonicity, we just insert y before deleting x.
- During the "consolidate" operation, a set X of keys are turned into a priority queue. Here we initiate the queue with a dummy $0 \le \min X$, which we remove after inserting the keys from X.

With these minor modifications, the construction from [12, §2.2] proves Lemma 1.1.